ENSEEIHT **3e année**
TLS-SEC **TP**

Vulnérabilités logicielles - TP3

14 novembre 2023

Ce TP a pour but de mettre en œuvre les classes d'attaques logicielles avancées vues en cours, le tout en présence de mécanismes de protection à l'état de l'art mis en œuvre par les systèmes d'exploitation GNU/linux.

Le système étudié met en œuvre un protocole de journalisation d'évènements transportés à l'aide d'un pipe nommé entre un client et un serveur. La mise en œuvre côté serveur du protocole de journalisation est vulnérable. Le serveur est privilégié et le client est sous maîtrise de l'attaquant.

L'étudiant devra exploiter cette vulnérabilité de façon à produire une primitive de lecture et d'écriture mémoire du processus serveur, contourner les mesures de protections modernes et enfin obtenir le recouvrement du processus attaqué par un shell (`execve("/bin/sh")`).

ATTENTION : La plupart des réponses aux questions posées dans le TP sont données après les questions pour permettre à tout le monde d'avancer. Afin de maximiser l'expérience pédagogique, veillez à bien parcourir linéairement le sujet.

---

1 Préparations

```
$ tar xvf lubuntu-18.04.15-tp-secu-logiciel-64.tgz
$ cd lubuntu-18.04.15-tp-secu-logiciel-64
# sudo si vous n'avez pas le droit de créer des interfaces et de changer kvm
$ ./start.sh
```

Une fois la VM lancée, rendez-vous dans le dossier `~n7/tp3`

Accès au fichiers de la VM

Qemu relaye en DNAT le couple `ip:port` hôte `localhost:2222` vers le couple de le couple `ip-vm:22` de la VM. Il est donc possible de se connecter sur le serveur OpenSSH de la VM pour télécharger les fichiers. Depuis l'hôte :

```
$ scp -P 2222 -r n7@localhost:/home/n7/tp3 .
$ sftp -P 2222 n7@localhost
```

2 Un protocole de journalisation (log) client / serveur

Ce TP étudie la mise en œuvre d'un protocole de log client / serveur vulnérable. Le protocole permet à un client unique d'envoyer un flux de données, préférablement texte, à un serveur qui va découper ce flux de texte en fichiers de taille définie par le client.

2.1 Description sommaire du protocole de log

Les fichiers créés par le serveur portent un nom de la forme : `logs/<majeur>.<mineur>.txt`. Le client contrôle directement la numérotation majeure des fichiers à l'aide de l'envoi de fin de fichier. Une fin de fichier ferme le fichier courant et ouvre un nouveau fichier de la forme : `logs/<majeur>+1.0.txt`. Lorsque la taille maximale du fichier courant est atteinte, le serveur ferme le fichier courant et ouvre un nouveau fichier de la forme : `logs/<majeur>.<mineur>+1.txt`. Le client contrôle la taille du fichier en envoyant des messages spéciaux (de contexte) au serveur qui vont spécifier une nouvelle taille maximale. La taille maximale supportée par le protocole est `0x100` octets par fichier. Dans le reste de ce document on appelle **session de log** les fichiers crées avec le même nombre majeur. Le client peut aussi demander au serveur d'inverser le contenu individuel des fichiers avec un message de contexte. Cette fonctionnalité est une coquille vide dans le code du serveur, elle sera utilisée seulement pour executer une de nos attaques.

Le client et le serveur communiquent à l'aide d'un canal de communication fiable en mode flux. Dans notre cas nous utilisons un pipe nommé (`mkfifo`). Ce protocole implique principalement deux primitives : des messages de données et des messages de contexte dont les entêtes sont décrites dans les tableaux 1 et 2. La dernière primitive (tableau 3) sert à terminer le processus serveur depuis le client.

L'image 1 illustre le scénario d'attaque que nous considerons. Un adversaire a corrumpu le client et peut y exécuter du code arbritraire. Dans ce TP nous simulons cette première attaque en modifiant directement le code du client.

Type	Nom	Description
`int32`	`type`	`0` : données
`int32`	`data_length`	taille des données placées après l'entête
`int32`	`end_of_packet`	fin de fichier majeur courant
`int32`	`-`	réservé

Table 1 – Entête de message de données

Nous remarquerons que les entêtes de message sont de la même taille pour simplifier leur traitement.

Le tableau 4 représente un paquet de données valide. La représentation des nombres sur le réseau est *Little Endian*, c'est à dire identique à celle des machines Intel (à bon entendeur...).

Type	Nom	Description
`int32`	`type`	`1` : contexte
`int32`	`data_length`	taille des données placées après l'entête
`int32`	`maximum_log_file_size`	taille maximum d'un fichier de log
`int32`	`reverse_order`	inverser les octets des fichiers de log

TABLE 2 – Entête de message de contexte

Type	Nom	Description
`int32`	`type`	`2` : fin de serveur
`int32`	`data_length`	taille des données placées après l'entête
`int32`	`-`	réservé
`int32`	`-`	réservé

TABLE 3 – Entête de message de fin de serveur

Un entête précède toujours des données à envoyer. La notation "C →" signifie que le client envoie un message au serveur. Dans le cadre de ce protocole, nous pouvons avoir l'interaction suivante :

1. C → `ctx` : taille des fichiers `0x20` octets
2. C → 8×`data` : `"Dummy\n"` : 8 lignes de log
3. C → `data.end_of_packet = 1` : client termine le fichier majeur en cours
4. C → `ctx` : nouvelle taille des fichiers `0x40` octets
5. C → 16×`data` : `"Dummy\n"` : 16 lignes de log
6. C → `data.end_of_packet = 1` : client termine le fichier majeur en cours
7. C → `end` : client termine le serveur

Côté serveur les premiers fichiers ne peuvent pas dépasser 0x20 octets. Si le prochain message reçu ne rentre pas, on ferme le fichier courant, on incrémente le numéro mineur et ouvre le prochain fichier. Dans le cas de notre exemple, les 8 premiers logs reçus (6 octets par message) sont découpés de la façon suivante :

— 1 fichier de 5×6 octets `"Dummy\n"`, appelé `0.0.txt`

— 1 fichier de 3×6 octets `"Dummy\n"`, appelé `0.1.txt`

Pour la deuxième session de log on a :

— 1 fichier de 10×6 octets `"Dummy\n"`, appelé `1.0.txt`

— 1 fichier de 6×6 octets `"Dummy\n"`, appelé `1.1.txt`

2.2 Mise en œuvre du protocole

Le client est décrit en langage C dans le fichier `client.c`, le serveur dans le fichier `server.c`. Un `Makefile` vous permet de générer le client et le serveur à l'aide

champ	Entête				Données	
	type	data_length	end_of_packet	réservé		
offset	00	04	08	0c	10	14
packet	00 00 00 00	06 00 00 00	00 00 00 00	00 00 00 00	'D' 'u' 'm' 'm' 'y' \n	

TABLE 4 – Message de données transportant un message de log de 6 caractères

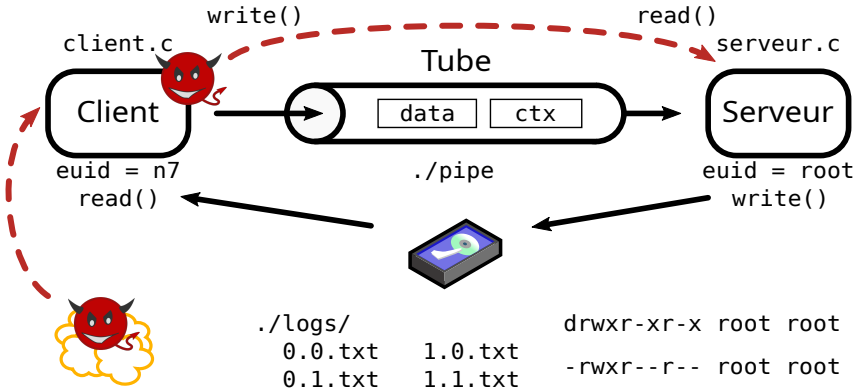

FIGURE 1 – Architecture client / serveur de log

du compilateur GCC : `make serve` et `make run_test`. Les recettes `dgb_serve` et `dbg_run_test` permettent de lancer respectivement GDB sur le serveur et le client pour les déboguer. La figure 1 décrit l'architecture de ce TP.

Le fichier `log_protocol.h` décrit les structures protocolaires du protocole de log. Le fichier `log.c` implémente les primitives de débogage et d'affichage d'information sur la sortie standard pour le client et le serveur.

Ouvrez le fichier `log.c` et observez la déclaration de la variable `FILE **file_debug;`. Dans quelle section du programme est-t-elle normalement déclarée (data / bss / text) ? Notez bien cette particularité pour plus tard.

2.3 Prise en main du code du client

Il est maintenant temps de prendre en main le code du client. Les fonctions `send_ctx()` et `send_data()` permettent d'envoyer au serveur des messages de contexte et de données logs.

Le programme client met en œuvre l'exemple de la section précédente à l'aide de la fonction `test1();`. Les fichiers de logs seront crées en cohérence.

Lisez le code de `test1()` ainsi que de des fonctions `send_ctx()` et `send_data()`.

2.4 Prise en main du code du serveur

Le serveur bufferise en mémoire les bouts de fichiers de logs reçus des messages de données. Il les écrit sur le disque une fois le buffer plein ou la fin de fichier reçue. La taille maximum du buffer est le minimum entre la taille définie par l'utilisateur et la taille maximum du buffer alloué en mémoire.

```
// Log file buffer management
struct log_file {
  char buf[_LOG_MAX_FILE_SIZE];
  [..]
  unsigned int minor;
  unsigned int major;
  unsigned int buf_size;
};
struct log_file log_file;
[..]
unsigned int maximum_log_file_size = _LOG_MAX_FILE_SIZE;
```

La structure du serveur `struct log_file` contient le buffer de logs `buf` ainsi que la quantité de données actuellement utilisée `buf_size`.

La variable `maximum_log_file_size` contient la taille maximum du buffer configurée par l'utilisateur avec un message de contexte. La macro `_LOG_MAX_FILE_SIZE` est la taille maximum configurable pour le buffer ainsi que sa taille réellement allouée (`0x100` octets). Enfin, les champs `minor` et `major` de la structure `log_file` permettent de mettre en œuvre la numérotation des noms de fichiers.

Dans quelle section est déclarée la variable `maximum_log_file_size` ? Est-ce la même que pour la structure `log_file` ?

2.5 Premiers développements

Il manque à l'implémentation actuelle du client la fonction permettant d'envoyer la troisième primitive protocolaire `end` pour fermer le serveur.

- En vous inspirant des fonctions `send_ctx()` et `send_data()`, écrivez une fonction C `send_end()` permettant d'envoyer un message de fin de serveur. Le prototype à respecter est le suivant : `void send_end(void);`
- Modifiez la fonction main et décommentez les fonctions nécessaires à la bonne terminaison du serveur à la fin des tests.

Testez vos modifications en exécutant le serveur, puis le client et comparez-les au listing suivant. Ouvrez un terminal pour l'exécution du serveur, un autre pour le client et un dernier pour afficher les fichiers de logs créés par exemple.

```
# Générer les cibles binaires
$ make
# Lancer le serveur
$ make serve
```

Le serveur se met en écoute sur le pipe nommé et attend des commandes du client. Attention, le serveur doit être lancé avant le client. Ouvrir un autre terminal et lancer le client :

```
$ make run_test
```

Le serveur termine correctement et les fichiers générés suivants sont :

```
$ cat logs/0.0.txt
Dummy
Dummy
Dummy
Dummy
Dummy
```

```
$ cat logs/0.1.txt
Dummy
Dummy
Dummy
```

```
$ cat logs/1.0.txt
Dummy
Dummy
Dummy
Dummy
Dummy
Dummy
Dummy
Dummy
Dummy
Dummy
```

```
$ cat logs/1.1.txt
Dummy
Dummy
Dummy
Dummy
Dummy
Dummy
```

3 Vulnérabilité de mise en œuvre du protocole

Le cœur de métier du serveur de log est placé dans la fonction `void run(void);` Une boucle de lecture effectue le traitement des messages du client en trois étapes :

— STEP I : Lecture d'un entête de message (`struct log_desc`)

— STEP II : Lecture des données suivant le descripteur s'il y en a.
NB : Cette lecture se fait dans un buffer temporaire appelé `temp_buf` de taille `_LOG_MAX_FILE_SIZE = 0x100`

— STEP III : Exécution de l'action associée au message
NB : Si message de données, on copie éventuellement `temp_buf` dans `log_file.buf`

Quand est-ce que le message de contexte peut-être envoyé ? Est-ce que le serveur s'assure que le client n'envoie pas de message de contexte au milieu d'une session de log ?

Nous allons par la suite tirer partie du laxisme du serveur vis à vis de cette interaction.

3.1 Vulnérabilité 1

Le serveur de log protège ses copies mémoire de plusieurs façons :

1. Chaque donnée reçue en STEP II ne peut faire plus que `_LOG_MAX_FILE_SIZE` octets, taille de `temp_buf`
2. Chaque copie EP III de `temp_buf` vers `log_file.buf` ne peut être plus grande que la taille restant dans `log_file.buf`, i.e. $\leq$ `maximum_log_file_size - log_file.buf_size`
3. Enfin, `maximum_log_file_size` ne peut être placé à une valeur $>$ `_LOG_MAX_FILE_SIZE`

De ce fait, toutes les tentatives de dépassements triviales ont l'air d'être impossibles ... **Mais est-ce suffisant ? ? ?**

Développez une fonction `test2()`, inspirée de `test1()`, qui va exécuter les actions protocolaires suivantes :

1. Fermer la session de log en cours avec l'envoi d'une fin de fichier
2. Placer la taille max de fichier de log à `0x20`
3. Envoyer 0x10 octets de log au serveur
4. Placer la taille max de fichier de log à `0x10`
5. Envoyer 0x1 octet de log au serveur
6. Fermer la session de log en cours avec l'envoi d'une fin de fichier
7. Éteindre le serveur

— Compilez et exécutez le client et la fonction `test2();`

— Que fait le serveur lorsque vous envoyez un octet de plus après avoir à la fois baissé la taille maximum à 16 et envoyé 16 octets auparavant ?

— Quelles sont les valeurs des variables `log_file.buf_size` et `maximum_log_file_size` après le traitement du deuxième paquet de données ?

— Identifiez la ligne de code C où le serveur calcule la place restante dans le buffer `log_file.buf`. Relevez la ligne où on affiche la place restante.

On voit qu'il est possible de modifier la taille maximum désirée du buffer côté serveur alors qu'on y a déjà placé des données.

- Modifiez la fonction `test2()` au niveau de l'étape 4, en remplaçant la taille maximum désirée par `0x0f`
- Compilez et exécutez le client et la fonction `test2();`
- Que fait le serveur lorsque vous envoyez un octet de plus après avoir à la fois baissé la taille maximum à 15 et envoyé seize octets auparavant ? Lisez la sortie du serveur commençant par `Remaining room in log file buffer:`
- Quelles sont les valeurs des variables `log_file.buf_size` et `maximum_log_file_size` après copie ?
- Quel type de vulnérabilité avez-vous déclenché ?

Nous sommes donc capables de dépasser `maximum_log_file_size`.

- Pouvons-nous aussi dépasser la taille allouée pour le buffer `_LOG_MAX_FILE_SIZE` ? Proposez au moins une méthode.
- Quelle est la quantité maximum de données que l'on peut copier en une fois dans `log_file.buf` en utilisant la vulnérabilité précédente ?

3.2 Exploitation de la vulnérabilité

Nous avons trouvé un buffer overflow dans la copie de `temp_buf` vers `log_file.buf`, il est temps de l'outiller.

- Écrire une fonction dans le client qui permet de remplir le tableau `log_file.buf` à `_LOG_MAX_FILE_SIZE` octets et de continuer d'écrire plus loin lors des prochains envois de données
- Voici l'algorithme à exécuter :
  1. Commencez par fermer (et donc flusher) la session de log courante avec un `end_of_file` pour replacer `log_file.buf_size` à zéro
  2. Placez ensuite la taille maximum à `_LOG_MAX_FILE_SIZE` avec un message de contexte
  3. Envoyez `_LOG_MAX_FILE_SIZE` données
  4. Repassez à `_LOG_MAX_FILE_SIZE`-1 avec un message de contexte
- Prototype de fonction : `void attack_set_integer_overflow(void);`

Cette fonction sera la brique de base de notre client pour construire des attaques plus complexes ! Il est temps de la tester.

- Écrivez une fonction `test3()` qui l'utilise, puis envoie un octet supplémentaire.
- Compilez et testez le programme client avec le serveur
- Constatez le dépassement d'un octet du tableau `log_file.buf`

4 Primitives de lecture / écriture

Nous avons trouvé un dépassement en écriture après le tableau `log_file.buf`. Il est temps maintenant de construire une réelle primitive d'écriture dans la mémoire du serveur. Nous essayerons dans un deuxième temps si possible d'en déduire une primitive de lecture de la mémoire du serveur.

4.1 Primitive d'écriture

Nous allons développer dans cette section une fonction qui permet d'écrire arbitrairement à la suite du tableau `log_file.buf` $n <$ `_LOG_MAX_FILE_SIZE` octets.

- Développez la primitive d'écriture qui écrit `size` octets après la fin de `log_file.buf` :
  `void attack_write_mem(char *buf, int size);`
- Algorithme à suivre :
  1. Préparer le dépassement de `log_file.buf` avec un appel à `attack_set_integer_overflow()`
  2. Envoyer le paquet contenant les données `temp_buf` de l'attaquant. Ne fermez surtout pas le fichier, c'est important pour la suite.

Nous allons tester cette fonction à l'aide des sections suivantes.

4.2 Accessibilité en écriture du dépassement

Nous pouvons dépasser arbitrairement en écriture avec `log_file.buf`. Que pouvons-nous écraser ?

Nous nous intéressons à ce qui est placé après le tableau `log_file.buf`. Commençons par les autres champs de cette structure :

Offset O_1^e : `unsigned int minor;`

Offset O_2^e : `unsigned int major;`

Offset O_3^e : `unsigned int buf_size;`

- Utilisez `gdb` sur `log_server` pour afficher l'emplacement mémoire de ces champs par rapport à la fin de `log_file.buf`
- Notez précisément l'offset des champs pour plus tard. Définissez des macro C, ex : `#define O_END_MINOR 0xcafe // offset Oe_1`

Nous pouvons écraser les champs `minor`, `major` et `buf_size`, placés respectivement aux offsets O_1^e, O_2^e et O_3^e. Nous utilisons cette propriété dans la prochaine section.

4.3 Nom de fichier arbitraire

Les fichiers de logs semblent être une source d'information intéressante pour développer une primitive de lecture mémoire si on arrive à contrôler ce que le serveur y écrit.

Une des difficultés que va devoir surmonter l'attaquant est de deviner et contrôler de façon déterministe quel sera le prochain fichier utilisé par le serveur. Les noms de fichiers sont générés à l'aide du compteur majeur et mineur de fichiers de la structure `log_file`.

Quel est le comportement du serveur vis à vis du fichier courant de sortie dans le dossier `logs/` lorsque le client envoie un message de fin de fichier ?

Si nous pouvons donc modifier le compteur majeur et provoquer une fin de fichier, nous pouvons influencer le nom des prochains fichiers utilisés par le serveur.

Si nous voulons placer le prochain nom de fichier à `logs/n.0.txt`, quelle est la valeur à affecter à`log_file.major` ?

Nous avons maintenant tous les éléments permettant le contrôle du prochain fichier ouvert depuis le client.

— Écrire une fonction qui va placer le prochain nom de fichier de logs selon une entrée choisie par l'attaquant :
`void attack_set_filename(int major);`

— Algorithme à suivre :

1. Écrire sur le serveur O_2^e+ 0x04 octets contenant : O_2^e octets de padding et le numéro `major` souhaité par l'attaquant -1
2. Fermer le fichier en cours avec une fin de fichier

Nous allons développer la fonction `test4()` pour évaluer cette fonctionnalité.

— Développez une fonction `test4()` qui

1. Place le nom de fichier à 42
   `attack_set_filename(42)`
2. Écrit `"Dummy\n"` dans le fichier
3. Ferme le fichier

— Exécutez les server avec gdb : `make dgb_serve`

— Exécutez le client

— Listez les fichiers pour confirmer la création du fichier `logs/42.0.txt`

Pour tester cette fonction de façon précise, il est nécessaire d'utiliser le débogueur `gdb` sur le serveur. Vous pouvez utiliser la cible make `dbg_serve` :
`$ make dbg_serve`.

4.4 Lecture dans un fichier de log à offset et taille donnés

Pour manipuler automatiquement les fichiers de logs en lecture, nous vous proposons la fonction au prototype suivant :

```
int read_file(int major, char *buf, int size, int offset);
```

Elle permet de lire dans un fichier appelé `logs/<major>.0.txt`, `size` octets, à offset `offset`, à destination de `buf`.

> Lisez tout de même rapidement le code de la fonction pour vous convaincre de la pertinence de l'approche.

Nous sommes maintenant capables de contrôler le fichier de sortie utilisé par le serveur pour écrire des données et d'y lire automatiquement à un offset et une taille donnée. Nous allons voir comment y écrire des données de la mémoire du serveur dans la prochaine section.

4.5 Primitive de lecture

La quantité de données écrites dans le fichier de log courant est contrôlée par le champ `log_file.buf_size` qui fait `32-bits`. Si nous maîtrisons à distance ce nombre, nous pouvons lire arbitrairement les données de la mémoire du serveur, à partir de `log_file.buf`.

> — Proposez une stratégie d'attaque qui utilise la primitive d'écriture mémoire afin de lire arbitrairement la mémoire du serveur placée après le tableau `log_file.buf`.
>
> — Proposez une formule F_1 pour calculer la taille à placer dans le champ `log_file.buf_size`.
>
> — Quelle action effectuée par le serveur faut-il considérer dans ce calcul ?

Il est maintenant temps de développer la primitive de lecture basée sur la primitive d'écriture.

— Développez une primitive de lecture au prototype suivant :
`void attack_read_mem_at(char *buf, int size, int offset);`
  — `buf` : buffer de sortie
  — `size` : taille du buffer de sortie
  — `offset` : offset de lecture
  ATTENTION : calculé à partir du DÉBUT du tableau, contrairement à la primitive d'écriture qui calcule à partir de la fin
— Algorithme à suivre :
  1. Placer le nom du fichier de sortie
  ex : `attack_set_filename(0x10);`
  2. Réécrire la taille du buffer à offset O_3^e en utilisant la primitive d'écriture et la formule F_1
  Attention, ceci incrémente le nom majeur du fichier de sortie
  ex : `attack_write_mem(&overflow[0], sizeof(overflow));`
  3. Fermer le fichier pour pousser les données dans le fichier de log
  4. Attention au temps que le serveur met pour créer le fichier sur le disque. Le client est le serveur ne sont pas synchrones !
  5. Extraire `size` données du fichier de log à offset `offset`
  ex : `read_file(0x10 + 1, buf, size, offset);`

5 Intrusion 1 : casser l'ASLR

Une des mesures modernes de tolérence aux intrusions est l'ASLR. Elle impose à un attaquant à agir à l'aveugle en terme d'espace d'adressage pour l'exécution d'une attaque.

Si nous prenons l'exemple d'un buffer overflow classique dans la pile, l'attaquant peut connaître de façon déterministe la distance du buffer dépassé avec l'emplacement de l'adresse de retour de la fonction vulnérable en mémoire. En outre, lorsqu'il s'agit d'écraser cette adresse en adressage absolu, il est impossible pour l'attaquant de savoir où est positionné du code utilisable. En théorie l'espace de recherche est de $2^{48-1-12}$ sous processeur intel en mode `ia-32e`.

Cependant, l'ASLR est loin d'être parfait dans sa mise en œuvre concrète. En effet, l'espace d'adressage virtuel est seulement décalé vis à vis d'un offset fixe tiré aléatoirement dans un intervalle restreint de l'espace d'adressage virtuel, la distance relative de chaque élément du programme n'étant pas modifiée. Par conséquent, connaître la position en mémoire après chargement du moindre élément d'un programme casse complètement l'ASLR. On est en mesure d'adresser relativement tous les éléments du programme vis à vis de celui-ci alors qu'on a son adresse absolue.

Le but de cette section est donc de lire une adresse absolue en mémoire après chargement du programme (un pointeur).

La primitive de lecture développée précédemment permet de lire arbitrairement

la mémoire à partir du début du tableau `log_file.buf`. Nous allons déterminer les éléments accessibles dans la section suivante.

5.1 Accessibilité en lecture du dépassement

Le programme `nm` permet d'inspecter les exécutables pour trouver dans quelle section les symboles sont déclarés et à quel offset de chargement du programme.
`$ nm log_server | grep <symbole>`
La section est indiquée par un caractère en majuscule ou en minuscule :

- `[bB]` : `bss`
- `[dD]` : `data`
- `[tT]` : `text`

Une minuscule signifie symbole local et une majuscule signifie symbole exporté, i.e. utilisable par d'autres programmes lors de l'édition des liens.

- Utilisez la commande `nm` pour trouver dans quelle section du programme se trouve la structure `log_file`
- Affichez ensuite tous les symboles de la section dans laquelle se place la structure `log_file`
  `$ nm log_server | grep -i ' <section-caracter> '`
- Triez les de façon à obtenir ceux qui sont placés après la structure `log_file`
- Quel pointeur déjà observé en section 1 est placé après ?
- Relevez précisément son offset par rapport au **début** de `log_file.buf` : O_1^b. Définissez la macro C correspondante : `#define O_START_FILE_DEBUG x`
- Quelle est la taille de ce type ?

Une analyse sémanique de la valeur du pointeur `@file_debug` permet de casser l'ALSR.

- Lisez `lst/log.c:14` et observez avec quoi la variable est initialisée. A quel symbole précédent, définit dans le `bss` le membre droit de l'affectation correspond-t-il ?
- Relevez son offset par rapport au début du programme : O_1^l. Définissez la macro C correspondante : `#define O_LOAD_STDOUT x`
- Proposez une formule F_2 permettant de calculer l'adresse de chargement du programme, B, à partir de la valeur de `file_debug` et O_1^l.

5.2 Lecture du pointeur pour "régler son compte à l'ASLR"

Il est temps de dérandomiser l'espace d'adressage à l'aide de la lecture du pointeur `FILE **file_debug`.

— Écrivez une fonction `test4()` qui lit le pointeur `file_debug` à l'offset O_1^b et calcule l'adresse de chargement B à l'aide de la formule F_2 et de O_1^l. Utilisez évidemment votre primitive de lecture !

— À l'aide de la commande `nm` relevez l'offset de la fonction `motd` par rapport au programme : O_2^l

— Enfin, à l'aide de O_2^l vous calculerez l'adresse dérandomisée de la fonction `motd`.

6 Intrusion 2 : casser le canarie de la fonction `reverse()`

La fonction `reverse()` implémente le champ de la primitive protocolaire `ctx.reverse_order`, c'est à dire inverser les caractères du buffer `log_file.buf` avant de les copier dans le fichier de log courant. Cette opération est déclenchée soit lors de la réception d'une fin de session (`end_of_file`), soit lorsque le buffer est plein.

La fonction `reverse()` n'est que partiellement mise en œuvre pour l'instant sur le code du serveur. Observez tout de même le code serveur de la fonction `reverse()`.

— Quelle opération critique est exécutée par le serveur ?

— Quelle vulnérabilité peut-être exécutée ?

— Dans quelle section mémoire ?

Les canaries sont une mesure de tolérance aux intrusions efficace contre les dépassements de buffer dans la pile qui visent la réécriture de l'adresse de retour d'une fonction (ici `reverse()`).

Les canaries sous GNU / linux ont la forme suivante : `0xXXXXXXXXXXXXXX00`. Le premier octet à zéro permet de stopper les buffer overflow exécutés avec une copie de chaîne (caractère de fin de chaîne). Les canaries des fonctions sont constants pour toutes les exécutions une fois qu'il ont été tirés au sort au démarrage du processus.

Pour réécrire l'adresse de retour de `reverse()` nous allons devoir deviner la valeur des 56 bits du canarie (C) et les offsets du canarie (O_4^e) et de l'adresse de retour (O_5^e) par rapport à la fin du buffer `temp_buf`.

6.1 Découvrir l'emplacement du canarie de `reverse()`

Dans cette section, nous allons relever les offsets de l'adresse de retour et du canarie par rapport à la fin du tableau `temp_buf`, destination de la copie vulnérable de la fonction `reverse()` pour réécrire son adresse de retour.

- Écrire une fonction `test6()` permettant de déclencher l'exécution de `reverse()`
- Algorithme :
  - Placez la taille maximum de fichier à 0x100 et le mode `reverse_order`
  - Envoyez `_LOG_MAX_FILE_SIZE` octets avec un message de données
  - Fermez le fichier (la session en cours)

Nous pouvons maintenant relever les offsets O_5^e et O_6^e à l'aide du débogueur. Le canarie est placé en général 16 octets avant l'adresse de retour.

- Lancez le serveur en mode de débogage avec la cible `make dbg_serve`, puis le client.
- Placez un point d'arret après le `memcpy()` de `reverse()`
  `(gdb) b 171`
- Exécutez le serveur et relevez les valeurs des offsets et créez les macros suivantes :
  ```
  #define O_END_COOKIE x // Offset O_e4
  #define O_END_RETURN x // Offset O_e5
  ```
  - `(gdb) p &temp_buf`
  - `(gdb) info frame`
  - `(gdb) x /60gx $rsp`

6.2 Exécution du buffer overflow dans la pile

Nous voulons maintenant que cette exécution déclenche un buffer overflow dans la pile de `reverse()`.

- Quel champ / variable contrôle la taille de la copie `memcpy()` ?
- Comment contrôler cette valeur depuis le client ?
- Proposez un algorithme qui permet de déclencher ce buffer overflow afin de réécrire présisément l'adresse de retour
- Combien de dépassements de buffers sont exécutés avec la même donnée dans le tableau `log_file.buf` ?
- Dessinez grossièrement la forme des données malveillantes à envoyer pour réécrire le canarie, l'adresse de retour et le contrôle de la taille. Positionnez les offsets dans le dessin !

6.3 La tolérance aux fautes qui dessert la tolérance aux intrusions ?

Dans un environnement classique, un écrasement non correct d'un canarie déclenche un exception qui est relayée à l'aide du signal `SIGABRT`, dont le traitement par défaut termine le programme : `SIG_DFL`.

Le serveur a été développé pour être tolérant aux fautes. Il surcharge notamment le traitement (*handler*) de signal associé à `SIGABRT`.

- Lisez les lignes de code associées aux appels de fonction `setjmp()` et `longjmp()`
- Que fait le serveur si un dépassement est détecté ?
- Comment pouvons nous tirer partie de ce mécanisme de tolérance aux fautes pour attaquer le canarie ?

6.4 Attaque du canarie

Il est maintenant temps de deviner la valeur C du canarie.

Quelle est dans notre cas la méthode la plus efficace en complexité pour attaquer le canarie ? Donnez sa complexité en notation asymptotique.

L'attaque par force brute ne peut réussir si et seulement si nous disposons d'un oracle nous permettant de deviner si la valeur choisie est correcte.

Quel est cet oracle dans notre cas ?

Nous avons une méthode d'attaque du canarie et un oracle, il est temps d'instrumenter cette attaque.

- Développez une fonction capable de bruteforcer un octet de canarie de la fonction `reverse()`. Son prototype doit être :
  `int attack_guess_stack_cookie(int stack_cookie_offset, char *guessed, int guessed_size);`
  - `stack_cookie_offset` : O_4^e
  - `guessed_buf` : tableau contenant les octets du cookie déjà résolus
  - `guessed_size` : la taille de ce tableau
- Développez une fonction `test7()` qui casse les 7 octets du canarie

Déboguez le code du serveur avec la cible de débogage `make dbg_serve`. Désactivez l'arrêt lors de la détection du signal `SIGABRT` pour ne pas ralentir l'attaque : `(gdb) handle SIGABRT nosto`

7 `Return in motd()` et *ROP shellcode*

Nous avons maintenant tous les paramètres pour exécuter du code réutilisé sur le serveur depuis un retour de la fonction `reverse()` :

- O_4^e : offset du canarie
- O_5^e : offset de l'adresse
- O_4^e : offset de la taille de la copie dans la pile : `file_log.buf_size`
- C : valeur du canarie

7.1 Return into x

Il est temps décrire une fonction qui réécrit l'adresse de retour avec une chaîne de *ROP* arbitraire.

Écrire la fonction suivante qui exécute sur le serveur une chaîne de *ROP* arbitraire. Voici le prototype à respecter :
`void attack_stack_bof_exec(int stack_cookie_offset, long int cookie, void *chain, int chain_size);`
- `stack_cookie_offset` : O_4^e
- `cookie` : valeur du canarie
- `chain` : chaîne de ROP copiée à 0x10 bytes après le canarie : O_5^e = adresse de retour
- `chgain_size` : taille de la chaîne de ROP

7.2 `Return in motd()`

Commencez par tester votre fonction `attack_stack_bof_exec()` avec une chaîne de *ROP* de taille 1 qui retourne dans le fonction `motd()`. Attention le programme va effectivement défaillir car le retour de `motd()` sera invalide.

Écrivez la fonction `test8()`.

7.3 *ROP shellcode*

Une fois que vous êtes sur du fonctionnement de la fonction `attack_stack_bof_exec()`, constituez une chaîne de *ROP* avec *ROPGadget* pour exécuter un véritable *shellcode* sur le serveur.

Écrivez la fonction `test9()` qui exécute un *ROP shellcode*.

8 Conclusion

Vous pouvez maintenant aller dormir ...

9 Références

`https://ctf101.org/binary-exploitation/stack-canaries/`